

## Unit 1.2 Analysis

Algorithms

EE/NTHU

Mar. 5, 2018

## Evaluating an Algorithm

- Some criteria to judge an algorithm
  - Does it do what we want it to do?
  - Does it work correctly according to the original specifications of the task?
  - Is there documentation that describes how to use it and how it works?
  - Are procedures created in such a way that they perform logical sub-functions?
  - Is the code readable?

### Definition 1.2.1. Space/Time complexity

The **space complexity** of an algorithm is the amount of memory it needs to run to completion. The **time complexity** of an algorithm is the amount of computer time it needs to run to completion.

- Performance evaluation can be divided into two phases:
  - Performance analysis: a priori estimates,
  - Performance measurement: a posteriori testing.

# Algorithm Examples

- Examples for space and time complexities analysis.

## Algorithm 1.2.2. Expression

```
1 Algorithm expr(x, y, z)
2 // Arithmetic expression.
3 {
4     return  $x + y + y * z + (x + y - z) / (x + y) + 4.0$ ;
5 }
```

- Given arguments  $x$ ,  $y$ , and  $z$ , the value of the expression is calculated.

# Algorithm Examples, II

- Both algorithms calculate the sum of the array  $A$  with  $n$  elements.

## Algorithm 1.2.3. Simple Sum

```
1 Algorithm Sum(A, n)
2 // Simple summation of n-element array  $A[1 : n]$ .
3 {
4     Sum := 0;
5     for i := 1 to n do
6         Sum := Sum +  $A[i]$ ;
7     return Sum;
8 }
```

## Algorithm 1.2.4. Recursive Sum

```
1 Algorithm RSum(A, n)
2 // Recursive summation of n-element array  $A[1 : n]$ .
3 {
4     if ( $n \leq 0$ ) then return 0; // Termination check.
5     else return  $A[n] + \text{RSum}(A, n - 1)$ ;
6 }
```

# Space Complexity

- The memory space needed for the preceding algorithms consists two parts:
  - A **fixed part** that is independent of the size of the problem.
    - Function instructions, constants, simple variables (such as indexing variables).
  - The **variable part** that depends on the particular problem.
    - Space for the referenced variables, recursion stack space, etc.
  - The **total space**  $S(P)$  for an algorithm  $P$  is

$$S(P) = c + S_P(\text{instance characteristic}). \quad (1.2.1)$$

where  $c$  is a constant.

- For Algorithm **Expression** the memory space needed are for variables  $x, y, z$ , and the result. Thus, no memory is needed that is specific to the instance of the problem, i.e.,  $S_P(\text{instance characteristic}) = 0$ .
- For Algorithm **Sum**,  $S_{\text{Sum}}(n) \geq (n + 3)$ .
  - $n$  for array  $A$ , and one for each variable:  $n, i$  and  $\text{Sum}$ .
- For Algorithm **RSum**,  $S_{\text{RSum}}(n) \geq 3(n + 1)$ .
  - Each recursive call needs to store formal parameters, local variables, and return address.
  - For this problem, it needs to store pointer to  $A$ ,  $n$  and the return address. (assume it takes 3 words)
  - The number of recursive calls is  $n + 1$ . Thus, total memory space needed is at least  $3(n + 1)$ .

# Time Complexity

- The **time complexity**  $T(P)$  of an algorithm is the time required to execute an algorithm.
  - In a general sense, the **compile time** should be included. But, the compile time does not depend on the size of the problem and, thus, is not the focus of the analysis.
  - The execution time should include all operations. Yet, this would make the analysis difficult.
- The time complexity is simplified to count the number of **program steps** when the algorithm execute,  $t_P$ .
  - In a loose sense, a program step is an expression.
- As in the following example, one can add an variable **count** to the algorithm **Sum** to count the number of program steps.
- From the example, the number of program steps for an array with  $n$  elements, the total number of program steps executed is  $2n + 3$ . Thus  $t_{\text{Sum}} = 2n + 3$ .

# Time Complexity, II

## Algorithm 1.2.5. Sum – Program Step Counting

```
1 Algorithm Sum(A, n)
2 // count is a global variable with initial value of 0.
3 {
4     Sum := 0;
5     count := count + 1; // for assignment
6     for i := 1 to n do {
7         count := count + 1; // for loop control
8         Sum := Sum + A[i];
9         count := count + 1; // for assignment
10    }
11    count := count + 1; // for loop termination
12    count := count + 1; // for return
13    return Sum;
14 }
```

- This algorithm is the same as Algorithm (1.2.3) but with lines 5, 7, 9, 11, 12 added
- After execution, global variable *count* has the number of program steps executed.

# Time Complexity, III

- The **RSum** algorithm can also be modified to count the number of program steps as the following page.
- The number of program steps for an array *A* with *n*,  $n > 0$ , elements is

$$t_{\text{RSum}}(n) = 2 + t_{\text{RSum}}(n - 1)$$

- Including the case of  $n = 0$ , we have the following recurrence relationship:

$$t_{\text{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0, \\ 2 + t_{\text{RSum}}(n - 1) & \text{if } n > 0. \end{cases}$$

- This recursive formula can be expanded for  $n > 0$  as

$$\begin{aligned} t_{\text{RSum}}(n) &= 2 + t_{\text{RSum}}(n - 1) \\ &= 2 + 2 + t_{\text{RSum}}(n - 2) \\ &\vdots \\ &= 2n + t_{\text{RSum}}(0) \\ &= 2n + 2 \end{aligned}$$

## Algorithm 1.2.6. RSum – Program Step Counting

```
1 Algorithm RSum( $A, n$ )
2 //  $count$  is a global variable with initial value of 0.
3 {
4      $count := count + 1$ ; // for if statement
5     if ( $n \leq 0$ ) then {
6          $count := count + 1$ ; // for return statement
7         return 0;
8     }
9     else {
10         $count := count + 1$ ; // for the expression and return statements
11        return  $A[n] + \text{RSum}(A, n - 1)$ ;
12    }
13 }
```

- This algorithm is the same as Algorithm (1.2.4) with lines 4, 6, 10 added.

# Time Complexity, V

## Definition 1.2.7. Input Size

The **input size** of a problem is defined to be the number of words (or the number of elements) needed to describe the instance of the problem.

- For the algorithm `Sum`( $A, n$ ) the input size is  $(n + 1)$ ,  $n$  for the number of elements of the array, and 1 for the value of  $n$ .
- The following algorithm adds two  $m \times n$  matrices,  $A$  and  $B$ , together to form a resulting matrix,  $C$ .

## Algorithm 1.2.8. Matrix Addition

```
1 Algorithm MAdd( $A, B, C, m, n$ )
2 //  $C[1 : m, 1 : n] := A[1 : m, 1 : n] + B[1 : m, 1 : n]$ , all are  $m \times n$  matrices.
3 {
4     for  $i := 1$  to  $m$  do
5         for  $j := 1$  to  $n$  do
6              $C[i, j] := A[i, j] + B[i, j]$ ;
7 }
```

# Time Complexity, VI

- Adding `count` to count the number of program steps as the following.

## Algorithm 1.2.9. Matrix Addition – Counting Steps

```
1 Algorithm MAdd(A, B, C, m, n)
2 // count is a global variable with 0 initial value.
3 {
4     for i:= 1 to m do {
5         count := count + 1; // loop-i control
6         for j:= 1 to n do {
7             count := count + 1; // loop-j control
8             C[i, j] := A[i, j] + B[i, j];
9             count := count + 1; // element addition
10        }
11        count := count + 1; // loop-j termination
12    }
13    count := count + 1; // loop-i termination
14 }
```

- Time complexity is  $2mn + 2m + 1$
- Input size is  $2mn + 2$

# Time Complexity – Table Approach

- An alternative approach to find algorithm complexity is the table approach
- For example

Statement	s/e	freq.	Total steps
1 Algorithm Sum (A, n)	0	—	0
2 // Simple summation.	0	—	0
3 {	0	—	0
4     Sum := 0;	1	1	1
5     for i:= 1 to n do	1	$n + 1$	$n + 1$
6         Sum := Sum + A[i];	1	$n$	$n$
7     return Sum;	1	1	1
8 }	0	—	0
Total			$2n + 3$

where `s/e` is step per execution,  
`freq.` is the frequency of execution.

- Algorithm `Sum(A, n)` has the time complexity of  $2n + 3$ .

# Table Approach, II

- RSum example

Statement	s/e	frequency		Total steps	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
1 Algorithm RSum ( $A, n$ )	0	—	—	0	0
2 // Recursive summation.					
3 {	0	—	—	0	0
4     if ( $n \leq 0$ ) then	1	1	1	1	1
5         return 0;	1	1	0	1	0
6     else return					
7 $A[n] + \text{RSum}(A, n - 1)$ ;	$1 + x$	0	1	0	$1 + x$
8 }	0	—	—	0	0
Total				2	$2 + x$

$$x = t_{\text{RSum}}(n - 1)$$

- Thus,

$$t_{\text{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0, \\ 2 + t_{\text{RSum}}(n - 1) & \text{if } n > 0. \end{cases}$$

# Table Approach, III

- MAdd example

Statement	s/e	freq.	total steps
1 Algorithm MAdd ( $A, B, C, m, n$ )	0	—	0
2 // $C := A + B$ , all are $m \times n$ matrices.			
3 {	0	—	0
4     for $i := 1$ to $m$ do	1	$m + 1$	$m + 1$
5         for $j := 1$ to $n$ do	1	$m(n + 1)$	$mn + m$
6 $C[i, j] := A[i, j] + B[i, j]$ ;	1	$mn$	$mn$
7 }	0	—	0
Total			$2mn + 2m + 1$

- Thus,  $t_{\text{MAdd}}(n) = 2mn + 2m + 1$ .

# Fibonacci Number

- Fibonacci number is defined as

$$f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2}, \quad n \geq 2. \quad (1.2.2)$$

- The following algorithm calculate  $f_n$  using iterative approach.

## Algorithm 1.2.10. Fibonacci

```

1 Algorithm Fibonacci(n)
2 // Compute the n-th Fibonacci number.
3 {
4     if (n ≤ 1) then return n; // f0 or f1, just return n.
5     else {
6         fim2 := 0; fim1 := 1; // fim2 = fi-2, fim1 = fi-1.
7         for i := 2 to n do {
8             fi := fim1 + fim2; // fi = fi-1 + fi-2.
9             fim2 := fim1; fim1 := fi; // Update fi-2 and fi-1.
10        }
11        return fi; // fn = fi.
12    }
13 }
```

# Fibonacci Number, II

Statement	s/e	frequency		Total steps	
		$n \leq 1$	$n \geq 2$	$n \leq 1$	$n \geq 2$
1 Algorithm Fibonacci (n)	0	—	—	0	0
2 // Compute the n-th Fibonacci number.					
3 {	0	—	—	0	0
4     if (n ≤ 1) then	1	1	1	1	1
5         return n;	1	1	0	1	0
6     else {	0	—	—	0	0
7         fim2 := 0; fim1 := 1;	2	0	1	0	2
8         for i := 2 to n do {	1	0	n	0	n
9             fi := fim1 + fim2;	1	0	n - 1	0	n - 1
10            fim2 = fim1; fim1 = fi;	2	0	n - 1	0	2n - 2
11        }	0	—	—	0	0
12        return fi;	1	0	1	0	1
13    }	0	—	—	0	0
14 }	0	—	—	0	0
Total				2	4n + 1

- Thus,

$$t_{\text{Fibonacci}} = \begin{cases} 2, & n \leq 1, \\ 4n + 1, & n \geq 2. \end{cases}$$



# Time Complexity

- The time complexity – the execution time – of an algorithm depends on the input.
  - Thus, it is usually expressed as a function of the input size.
  - It can be expressed as a function of part of the input size.
    - For example,  $t_{\text{MAdd}}$  as a function of  $m$ , number of rows, only.
    - If such complexity is of interest to a user.
- In evaluating the time complexity of an algorithm, the number of steps is not well defined.
  - It can be a simple comparison, an addition, a multiplication, or even a complex expression.
  - Thus, the exact number is not very important.
  - The growth of the time complexity as the input size grows is usually of more interest.
- The asymptotic complexity will be studied more later.

## Amortized Analysis

- In C, array size is fixed. To handle data without prior knowledge of its size, dynamically allocated array should be used.

### Algorithm 1.2.11.

```
1 Algorithm Dynamic_Store( $A, size, index, item$ )
2 // Store item into a dynamic array  $A$  of  $size$ .
3 {
4     if ( $size = 0$ ) then { // Initial call.
5          $size := 1$ ;  $A := \text{malloc}(size \times \text{sizeof}(typeA))$ ;
6     }
7     else if ( $index > size$ ) then { // Array  $A$  is full. Double  $A$ .
8          $size := 2 \times size$ ;
9          $B := \text{malloc}(size \times \text{sizeof}(typeA))$ ;
10        for  $i := 1$  to  $index - 1$  do  $B[i] := A[i]$ ; // Copy old data.
11        free( $A$ );
12         $A := B$ ; // Pointer assignment.
13    }
14     $A[index] := item$ ; // Store into array  $A$ .
15     $index := index + 1$ ;
16 }
```

# Amortized Analysis, II

- All function parameters are assumed to be called by reference.
- Before the first call to `Dynamic_Store` algorithm, variable *size* should be initialized to 0 and *index* to 1.
- When the algorithm is called, one array storage operation is needed most of the time.
  - In this case, the complexity is  $\Theta(1)$ .
- However, when  $index = 2^k + 1, k = 0, 1, 2, \dots$ , then  $2^k + 1$  array storage operations are needed.
  - Let  $n = index$ , in this case,  $n$  operations are needed.
  - The complexity is  $\Theta(n)$ .
- Overall complexity is  $\mathcal{O}(n)$ .
- This overestimated the time complexity.
- **Amortized analysis** should be used for tighter bound. Three methods available:
  - **Aggregate analysis**
  - **Accounting method**
  - **Potential method**

## Aggregate Analysis

- The **aggregate analysis** performs the algorithm  $n$  times to get  $T(n)$  operations, then the average performance of the algorithm is then  $T(n)/n$ .
- For the `Dynamic_Store`( $A, size, index, item$ ) algorithm the cost of  $index = i, c_i$  is

$$c_i = \begin{cases} i & \text{if } i = 2^k + 1, k \in \mathbb{N}, \\ 1 & \text{otherwise.} \end{cases} \quad (1.2.3)$$

<i>index</i>	1	2	3	4	5	6	7	8	9
<i>size</i>	1	2	4	4	8	8	8	8	16
$c_i$	1	2	3	1	5	1	1	1	9
$\sum c_i$	1	3	6	7	12	13	14	15	24

- Total cost for  $n$  `Dynamic_Store` calls is

$$T(n) = \sum_{i=1}^n c_i \leq n + \sum_{j=1}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n. \quad (1.2.4)$$

- Thus, the amortized cost of a single call is 3.
- The amortized complexity of the algorithm is  $\mathcal{O}(1)$ .

# The Accounting Method

- The amortized analysis performs a sequence of  $n$  calls of the algorithm to find the average cost.
- The **actual cost**  $c_i$  of the algorithm may vary for different instance  $i$ .
- The **amortized cost**  $\hat{c}_i$  can be anything but to approach the actual cost over  $n$  calls, the following relationship must hold for all  $n > 0$ .

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i. \quad (1.2.5)$$

- The accounting method is then to select a amortized cost  $\hat{c}_i$  and show that Eq. (1.2.5) holds.
  - The smaller  $\left( \sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i \right)$  the more accurate amortized cost is.

## The Accounting Method, II

- For the `Dynamic_Store` algorithm example
- Choose  $\hat{c}_i = 3$ , we have

<i>index</i>	1	2	3	4	5	6	7	8	9
<i>size</i>	1	2	4	4	8	8	8	8	16
$c_i$	1	2	3	1	5	1	1	1	9
$\sum c_i$	1	3	6	7	12	13	14	15	24
$\hat{c}_i$	3	3	3	3	3	3	3	3	3
$\sum \hat{c}_i$	3	6	9	12	15	18	21	24	27
$\sum \hat{c}_i - \sum c_i$	2	3	3	5	3	5	7	9	3

- When  $\sum \hat{c}_i - \sum c_i > 0$ , we have net credits for future operations.
- It can be shown that  $\sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i \geq 3$  for all  $n \geq 2$ .
- Thus, the amortize cost per operation is 3 and the amortized complexity is  $\mathcal{O}(1)$ .

# The Potential Method

- The potential method associates a non-negative potential function,  $\Phi_i$ , with the  $i$ -th operation of the algorithm such that

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \quad (1.2.6)$$

The amortized cost at the  $i$ -th operation is the actual cost plus the potential difference between those two operation.

- The potential function represents the energy barrier for each operation.
- Thus,

$$\begin{aligned} \sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi_i - \Phi_{i-1}) \\ &= \sum_{i=1}^n c_i + \Phi_n - \Phi_{n-1} + \Phi_{n-1} - \Phi_{n-2} \cdots + \Phi_1 - \Phi_0 \\ &= \sum_{i=1}^n c_i + \Phi_n - \Phi_0 \end{aligned} \quad (1.2.7)$$

## The Potential Method, II

- Note that Eq. (1.2.5) still needs to be satisfied.
- Thus,

$$\Phi_n \geq \Phi_0, \quad \text{for all } n \geq 1. \quad (1.2.8)$$

where  $\Phi_0$  can be chosen arbitrarily, and is usually set to be 0.

- Again, the average amortized cost represents the amortized complexity of the algorithm.
- Take the **Dynamic\_Store** algorithm as an example, note that  $index > size/2$ , thus we can choose the following potential function.

$$\Phi_i = 2i - size_i, \quad (1.2.9)$$

where  $i = index$  and  $\Phi_0 = 0$ .

<i>index</i>	1	2	3	4	5	6	7	8	9
<i>size</i>	1	2	4	4	8	8	8	8	16
$c_i$	1	2	3	1	5	1	1	1	9
$\Phi_i$	1	2	2	4	2	4	6	8	2
$\hat{c}_i$	3	3	3	3	3	3	3	3	3

- Note that  $\Phi_i \geq 0$ .

- In the case that  $index \leq size$  no `malloc` is needed and  $size_i = size_{i-1}$ .

$$\begin{aligned}\widehat{c}_i &= c_i + 2i - size_i - 2(i-1) + size_{i-1} \\ &= 1 + 2i - 2i + 2 = 3.\end{aligned}\tag{1.2.10}$$

Note that  $c_i$  is given in Eq. (1.2.3).

- In the case that  $index > size$  when calling `Dynamic_Store` we have  $size_i = 2 \times size_{i-1} = 2(i-1)$ .

$$\begin{aligned}\widehat{c}_i &= c_i + 2i - size_i - 2(i-1) + size_{i-1} \\ &= i + 2i - 2i + 2 - 2i + 2 + i - 1 = 3.\end{aligned}\tag{1.2.11}$$

- Thus, we have the amortized cost per operation is  $\widehat{c}_i = 3$ .
- The amortized complexity of the algorithm is  $\mathcal{O}(1)$ .

## Binary Counter

- The  $m$ -bit incrementing binary counter algorithm is shown below.

### Algorithm 1.2.12.

```
1 Algorithm BinCount( $m, D$ )
2 // Increment  $m$ -bit binary array  $D[m-1:0]$ .
3 {
4      $i := 0$ ;
5     while ( $i < m$  and  $D[i] = 1$ ) do {
6          $D[i] := 0$ ;
7          $i := i + 1$ ;
8     }
9     if ( $i < m$ ) then  $D[i] := 1$ ;
10 }
```

- In this algorithm, the `while` loop on lines 5-8 determines the cost of the operation, but it is not a constant.
- Worst-case complexity is  $\mathcal{O}(m)$  due to the `while` loop on lines 5-8).
- How about the average-case complexity?

## Binary Counter – Example

- Example of  $\text{BinCount}(m, D)$  partial execution result with  $m = 5$  is shown below (Assuming  $D[m - 1 : 0]$  are all 0's initially.)

$D[4]$	$D[3]$	$D[2]$	$D[1]$	$D[0]$	$c_i$	$\sum c_i$
0	0	0	0	1	1	1
0	0	0	1	0	2	3
0	0	0	1	1	1	4
0	0	1	0	0	3	7
0	0	1	0	1	1	8
0	0	1	1	0	2	10
0	0	1	1	1	1	11
0	1	0	0	0	4	15
0	1	0	0	1	1	16
0	1	0	1	0	2	18
0	1	0	1	1	1	19
0	1	1	0	0	3	22
0	1	1	0	1	1	23
0	1	1	1	0	2	25
0	1	1	1	1	1	26
1	0	0	0	0	5	31

## Binary Counter – Aggregate Analysis

- Let the number of bits that change states be the cost of operation,  $c_i$ .
- The aggregate analysis execute the algorithm  $n$  times to find the total cost of operation and then the average can be found.
- Note that bit  $D[0]$  changes state on every call.
- Bit  $D[1]$  changes state every other time.
- Bit  $D[2]$  changes state every fourth time.
- Hence, we have

$$\sum_{i=1}^n c_i = n + n/2 + n/4 + \dots + n/2^m < 2n. \quad (1.2.12)$$

- Thus, the total amortized cost is  $\mathcal{O}(n)$
- And the amortized cost per operation is  $\mathcal{O}(1)$ .

# Binary Counter – Accounting Method

- In accounting method, we need find  $\hat{c}_i$  that satisfies Eq. (1.2.5).

$D[4]$	$D[3]$	$D[2]$	$D[1]$	$D[0]$	$c_i$	$\sum c_i$	$\hat{c}_i$	$\sum \hat{c}_i$
0	0	0	0	1	1	1	2	2
0	0	0	1	0	2	3	2	4
0	0	0	1	1	1	4	2	6
0	0	1	0	0	3	7	2	8
0	0	1	0	1	1	8	2	10
0	0	1	1	0	2	10	2	12
0	0	1	1	1	1	11	2	14
0	1	0	0	0	4	15	2	16
0	1	0	0	1	1	16	2	18
0	1	0	1	0	2	18	2	20
0	1	0	1	1	1	19	2	22
0	1	1	0	0	3	22	2	24
0	1	1	0	1	1	23	2	26
0	1	1	1	0	2	25	2	28
0	1	1	1	1	1	26	2	30
1	0	0	0	0	5	31	2	32

- $\hat{c}_i = 2$  is a choice and the amortized cost per operation is  $\mathcal{O}(1)$ .

# Binary Counter – Potential Method

- In potential method, we need to find the potential function that satisfies Eqs. (1.2.7) and (1.2.8), then the amortized cost can be found using Eq. (1.2.9).
- Define the potential function as

$$\Phi_i = \sum_{i=0}^{m-1} D[i]. \tag{1.2.13}$$

That is  $\Phi_i$  is the number of set bits ( $D[i] = 1$ ).

- Let  $r_i$  be the number of bits reset to 0 for the  $i$ -th operation, then

$$c_i = r_i + 1. \tag{1.2.14}$$

- Note that  $r_i$  is simply the number iteration for the **while** loop on lines 5-8 of Algorithm (1.2.12), and the extra 1 comes from line 9.
- Thus for the  $i$ -th operation,

$$\Phi_i = \Phi_{i-1} - r_i + 1. \tag{1.2.15}$$

- And

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = r_i + 1 + \Phi_{i-1} - r_i + 1 - \Phi_{i-1} = 2. \tag{1.2.16}$$

- Thus, the amortized cost per operation is  $\mathcal{O}(1)$ .

# Binary Counter – Potential Method, II

- Potential method in 5-bit binary counter example.

$D[4]$	$D[3]$	$D[2]$	$D[1]$	$D[0]$	$c_i$	$\Phi_i$	$\hat{c}_i$
0	0	0	0	1	1	1	2
0	0	0	1	0	2	1	2
0	0	0	1	1	1	2	2
0	0	1	0	0	3	1	2
0	0	1	0	1	1	2	2
0	0	1	1	0	2	2	2
0	0	1	1	1	1	3	2
0	1	0	0	0	4	1	2
0	1	0	0	1	1	2	2
0	1	0	1	0	2	2	2
0	1	0	1	1	1	3	2
0	1	1	0	0	3	2	2
0	1	1	0	1	1	3	2
0	1	1	1	0	2	3	2
0	1	1	1	1	1	4	2
1	0	0	0	0	5	1	2

## Amortized Analysis

- In amortized analysis a sequence of  $n$  operations are performed to find the worst-case total operations.
- The time complexity of a single operation is then the total operation cost divided by the number of operation,  $n$ .
- Three methods are available:
  - Aggregate analysis,
    - More systematic.
  - Accounting method,
    - Usually the amortized cost is assumed and proven to be correct.
  - Potential method,
    - Need to find the potential function.
    - A tool to prove the amortized cost.



- Space and time complexities
- Algorithm examples
- Time complexity
  - Counting number of steps
  - Table approach.
- Amortized analysis
  - Aggregate analysis
  - Accounting method
  - Potential method

